Research Statement – Anvesh Komuravelli

My research interest is in building scalable tools and techniques for developing programs in a *scientific* manner. The question of how to develop software has been around since the very beginning of computer science but, its ever growing complexity and diversity continue to pose challenges. Given our increasing dependency on automation, the concern for correctness is no longer limited to "safety-critical" systems like an airplane but also applies to systems that are intended to be secure and ensure privacy like the apps on our smartphones and tablets. Even the simplest of the bugs like out-of-bounds array access can go unnoticed for many years as we have recently seen with the famous Heartbleed bug. Moreover, present day programming is very iterative with adding new features, adapting to new platforms, improving the efficiency, etc. which introduces new challenges in terms of good abstract representations for code readability and maintainability, some of the essential considerations from a Software Engineering perspective. I am interested in *exploring the scientific challenges in computer programming and coming up with reliable and effective solutions.*

My research focus so far has been on *developing and utilizing automatic verification techniques for program correctness.* Program correctness has received much attention from early on by great minds such as Dijkstra, Floyd, and Hoare, to name a few. The idea of *deductive verification*, a framework for formal guarantees using mathematical logic, came into being with the desire of ensuring utmost correctness of programs. But, deduction was primarily a manual and tedious task, making it very challenging to be applied to realistic systems. However, the advent of *automatic* verification techniques in the last few decades (which are necessarily incomplete due to undecidability) was a game changer: such techniques can now automatically produce deductive proofs of program correctness.

Nevertheless, with the increasing complexity of programs, *scalability* remains a key challenge in making verification tools practical. My Ph.D. research, in particular, tries to address several fundamental limitations in present day verification techniques by developing algorithms that (a) efficiently utilize logical decision procedures, (b) incorporate new *abstract reasoning* and *active learning* techniques, and (c) exploit the inherent modularity in the program/system under verification, the kind of reasoning absent in existing techniques. My work has resulted in verification algorithms that are significantly better than the state-of-the-art, both in theory and in practice.

I will now briefly elaborate on my research experience [1], followed by my future research interests.

Exploiting Modularity in Procedural Programs. The last decade has witnessed a breakthrough in automatic verification with SAT(isfiability)-based methods, thanks to the efficient SAT and SMT (SAT Modulo logical Theories to handle arithmetic, etc.) solvers that exist today. In an SMT-based framework for checking safety specifications, the verification task is divided into multiple subproblems, each of which is concerned with a well-defined subset of all possible executions of the program, e.g., a fixed number of iterations of every loop and a fixed depth of procedure recursion. For many useful cases, each such subproblem is actually decidable and reduces to SMT. However, these methods suffer from exponentially growing SMT problems that need to be solved, even for sequential programs, and existing methods have only addressed the problem heuristically.

The key idea in my work [7] for addressing the above exponential blow-up is to exploit the procedure-level modularity inherent in sequential programs by performing verification in a *compositional* manner. That is, the algorithm analyzes individual procedures, as opposed to the whole program at once, by maintaining and utilizing logical formulas that *under-* and *over-approximate* the behavior of the procedures being called. The two kinds of approximations enable reuse and help avoid potential re-computations. However, the approximations can have implicitly bound auxiliary variables which can grow exponentially if we are not careful. To this end, my work also introduces a new technique called *Model Based Projection* (MBP) for obtaining cheap approximations to eliminating the auxiliary variables (lazy Quantifier Elimination) without giving up on completeness.

The resulting compositional algorithm is polynomial-time, a significant improvement over the state-of-the-art. Moreover, unlike existing compositional *heuristic* algorithms, the technique proposed by

[1] I acknowledge the invaluable contributions of my PhD advisor and other collaborators to my research experience.

my work is guaranteed to terminate for each subproblem of the verification task. I have implemented this algorithm in our verification tool SPACER [1], built using the infrastructure of Z3 [4], one of the best industrial-strength SMT solvers we have today, developed by Microsoft Research (MSR). In fact, it is my impression that Z3 is being deployed in several development projects at Microsoft. On many practical benchmarks from the 2014 International Competition on Software Verification (SV-COMP 2014) [3], SPACER *outperforms Z3 by orders of magnitude* and *MSR plans to incorporate the ideas of* SPACER *into Z3*.[2] Moreover, the ideas introduced by the MBP technique for efficiently approximating Quantifier Elimination can be also used (and are being used in my ongoing work) in other problems such as approximating the image computation in Abstract Interpretation, and analyzing logical formulas with alternating quantifiers that arise in automatic synthesis techniques.

Proof-Guided Analysis. While compositionality can be used to efficiently solve a subproblem of the verification task, the task itself is undecidable, in general. Existing SMT-based algorithms work by obtaining *proofs* of the subproblems and heuristically trying to *generalize* them to a safety proof of the entire program (verification task). However, it can be quite challenging to *guide* the generalizations in the desired way and we observed that even for simple programs, the state-of-the-art techniques can have difficulty in converging to a safety proof.

To address the problem with convergence, my work [8] proposes to use an *abstraction* of the program (e.g., one can abstract an *increment by 1* operation to an *increment by a positive number* operation) to obtain proofs of the subproblems, as such proofs tend to be more general and hence, more likely to converge. To automatically infer useful abstractions, the algorithm uses proofs of the previous subproblems themselves and discovers what details may be relevant for a global proof (*Proof-Based Abstraction*). When the abstractions are too coarse to be safe, it uses a *counterexample-based* strategy to refine them (*CounterExample Guided Abstraction Refinement*). Using the implementation of the algorithm in our tool SPACER, we have shown a *significant improvement over state-of-the-art on benchmarks from the 2013 International Competition on Software Verification* (SV-COMP 2013) [2].

The abstractions computed automatically by SPACER can have other advantages, such as, comparing different versions of the same software, avoiding potential recomputations in verifying a modified version of the program, kick-starting a different analysis like bit-precise reasoning, etc. The SPACER tool [1], which combines ideas from both the above techniques, is in fact an efficient SMT solver for *second-order logical formulas* and the case of program verification is only a specific instance corresponding to checking if *there exists a correctness proof that satisfies the verification conditions*.

Assume-Guarantee Reasoning. The above mentioned techniques do not generalize to the case of concurrent behavior in programs. This is because, the number of states of a concurrent system can grow exponentially in the number of components (the infamous *state-space explosion* problem), and hence, even a polynomial (in the number of states) time algorithm is no good. Moreover, there are many concurrent systems where probabilistic behavior is inherent as well, such as distributed randomized network protocols, security protocols, models of biological systems, etc. However, present day algorithms for such systems are monolithic and analyze the entire system at once.

To address state-space explosion for *finite-state* concurrent, probabilistic programs, my work proposes the first *compositional reasoning* techniques for checking safety. The techniques employ an *assume-guarantee* style reasoning, where each component is analyzed separately, by making use of a *small* environmental assumption about the rest of the components. To automatically synthesize *sufficiently small* assumptions that avoid state-space explosion, my work [10] describes an *active learning* framework where conjectures are candidate probabilistic systems for the assumptions and counterexamples happen to be *tree-shaped* probabilistic systems. The framework proposes new ideas for state-space partitioning and can be of independent interest. I also developed an alternative solution [9] for inferring environmental assumptions by adapting a counterexample-based strategy mentioned earlier. Our

[2] Personal communication with Nikolaj Bjørner, co-developer of Z3.

implementation of this algorithm has shown *significant gains in checking safety of randomized network protocols with multiple, concurrent components over monolithic techniques.*

Future Outlook

I believe that the existing process of posteriori correctness checking, after the program is developed, comes too late into the picture of program development and is not entirely satisfactory. On the other hand, automatic synthesis techniques are hard to scale as they need considerably richer specification logics to capture the necessary details while many useful logics are undecidable (e.g., first-order logic) and logical formulas tend to easily become incomprehensible. Furthermore, there are issues beyond correctness (e.g., code readability) that are not captured by present day synthesis or verification technology. For these reasons, I am interested in exploring alternative scientific methods.

Few decades ago, Deductive Reasoning came into existence side-by-side with Artificial Intelligence, with the former resulting in breakthroughs in automated reasoning (e.g., *resolution*). I envision a day, not far from today, when the paths merge again. I believe that one can leverage the advances in *Human Computer Interaction* to develop intuitive specification formalisms (e.g., spatial or diagrammatic, speech and text in natural language), which also raises the level of abstraction of programming and can help with readability. I also believe that one can utilize techniques from *Natural Language Processing* and related areas to interpret and translate/manipulate the specifications and techniques from *Machine Learning* to aid program synthesis. Such a scientific framework can also pave the way for a divide-and-conquer approach making synthesis and the final verification step simpler tasks.

In the short run, as automatic verification techniques will remain an integral part of the development process, I want to continue exploring questions such as the following.

Verification + Synthesis. I am very interested in exploring the *synergy* between program verification and its flip-side: program *synthesis* from specifications. I will give two motivating examples for this. (a) Manual deductive verification often utilizes auxiliary *ghost* code to express complex relationships between program variables such that normal program execution is unaffected. A simple example is to count the number of iterations of a loop. I am beginning to explore the question of how to *automatically* synthesize such ghost code to aid verification. (b) Present day scalable verification techniques are mostly catered towards checking *shallow* specifications, e.g., ones that have a significant Boolean structure and simple arithmetic. I am currently exploring how to use such scalable techniques for verifying a richer class of specifications with universal quantifiers, by synthesizing sufficient quantifier instantiations. Moreover, given the rise in domain-specific automatic synthesis techniques (e.g., strings in Spreadsheets [5], data-structures from relational representations [6]), I believe that a synergy between the two approaches can take us quite far in terms of scalability. For example, the domain-specific synthesizer can utilize the (shallow) facts of the rest of the program inferred by the verifier.

Efficient Probabilistic Analysis. With the growing complexity of software, a practical alternative for scaling verification technology is to focus on *probabilistic correctness guarantees* as opposed to ensuring correctness in all scenarios. However, such techniques are significantly under-developed when compared to non-probabilistic analyses. As I mentioned earlier, my work is the first to develop algorithms for a complete compositional analysis framework for *finite-state* probabilistic systems. However, real software deals with infinite data types (or practically infinite, if physical limitations of the machine are taken into account). To develop practical probabilistic analysis techniques, a lot of foundational work needs to be done, such as, inventing expressive logics for proofs, appropriate proof-systems, algorithms for constraint solving in presence of probabilities (finding suitable problem formulations as well as solutions), etc. I believe that such probabilistic analyses also find interesting applications in other domains, e.g., *probabilistic programming* in the Machine Learning community.

References

[1] SPACER Tool. `http://spacer.bitbucket.org/`.

[2] Software Verification Competition. Part of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), 2013. `http://sv-comp.sosy-lab.org/`.

[3] Software Verification Competition. Part of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), 2014. `http://sv-comp.sosy-lab.org/`.

[4] L. de Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2008.

[5] S. Gulwani, W. R. Harris, and R. Singh. Spreadsheet Data Manipulation using Examples. *Commun. ACM*, 55(8):97–105, 2012.

[6] P. Hawkins. *Data Representation Synthesis*. PhD thesis, Stanford University, 2012.

[7] A. Komuravelli, A. Gurfinkel, and S. Chaki. SMT-Based Model Checking for Recursive Programs. In *Proceedings of the 26th International Conference on Computer Aided Verification (CAV)*, 2014.

[8] A. Komuravelli, A. Gurfinkel, S. Chaki, and E. M. Clarke. Automatic Abstraction in SMT-based Unbounded Software Model Checking. In *Proceedings of the 25th International Conference on Computer Aided Verification (CAV)*, 2013.

[9] A. Komuravelli, C. S. Păsăreanu, and E. M. Clarke. Assume-Guarantee Abstraction Refinement for Probabilistic Systems. In *Proceedings of the 24th International Conference on Computer Aided Verification (CAV)*, 2012.

[10] A. Komuravelli, C. S. Păsăreanu, and E. M. Clarke. Learning Probabilistic Systems from Tree Samples. In *Proceedings of the 27th International Symposium on Logic in Computer Science (LICS)*, 2012.